UNCLASSIFIED

AD 296 186

Reproduced
by the

ARMED SERVICES TECHNICAL INFORMATION AGENCY
ARLINGTON HALL STATION
ARLINGTON 12, VIRGINIA

UNCLASSIFIED

NOTICE: When government or other drawings, specifications or other data are used for any purpose other than in connection with a definitely related government procurement operation, the U. S. Government thereby incurs no responsibility, nor any obligation whatsoever; and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use or sell any patented invention that may in any way be related thereto.

CATALOGED BY ASTIA
AS AD No. 296186

63-2-4

UNCLASSIFIED

296 186

ASTIA
FEB 15 1963

COMPUTER ASSOCIATES, INCORPORATED

FORTY-FOUR WINN STREET, WOBURN, MASS.

Best
Available
Copy

"Requests for additional copies by Agencies of the Department of Defense, their contractors, and other Government agencies should be directed to the:

ARMED SERVICES TECHNICAL INFORMATION AGENCY
ARLINGTON HALL STATION
ARLINGTON 12, VIRGINIA

Department of Defense contractors must be established for ASTIA services or have their 'need-to-know' certified by the cognizant military agency of their project or contract."

"All other persons and organizations should apply to the:

U.S. DEPARTMENT OF COMMERCE
OFFICE OF TECHNICAL SERVICES
WASHINGTON 25, D.C.

COMPUTER ASSOCIATES, Inc. / Forty-Four Winn Street • Woburn, Massachusetts • WEllo 5-2121

SUMMARY OF A METHOD FOR THE AUTOMATIC CONSTRUCTION OF SYNTAX DIRECTED COMPILERS

by
Stephen Warshall
Computer Associates, Inc.
44 Winn Street
Woburn, Massachusetts

Contract No. AF19(628)-419
Project No. 4641, Task 464102

Scientific Report No. 2
December, 1962
Report No. AFCRL-62-955

Electronics Research Directorate
Air Force Cambridge Research Laboratories
Office of Aerospace Research
United States Air Force
Bedford, Massachusetts

This report describes a recently developed technique for the rapid realization of compilers for new source languages and new target machines.

The method is built around a table-driven translation algorithm, which was announced some time ago*. This algorithm is embodied in a small general-purpose (language- and machine-independent) program, called the "Translater", which accepts strings of characters, syntactically analyzes them, and emits pseudo-code macro-instructions, performing arbitrarily complex investigations of syntactic context to decide this emission.

The Translater is particularized to a given source-target pair by a set of tables which define the syntax of the source language and the strategy to be used in studying context and emitting macros. The newly-developed technique consists of the creation of a readable, formal language for describing both the syntax and the strategy of macro generation; thus, the Translater can, by translating a message in this new language, create an instance of its own tables.

This capability of the Translater is formally trivial and would be of little interest, were it not that:

1. That portion of the language which defines syntax corresponds very closely with the "Backus normal form" and can be transliterated almost directly from any conventional formal description of syntax.
2. That portion of the language which deals with the strategy of macro emission is fairly powerful and rather surprisingly readable.

In the sequel, we first sketch the Translater algorithm sufficiently well to introduce the critical control tables. Then we describe the gross

---

*S. Warshall, "A Syntax Directed Generator", Proceedings of the EJCC, 1961, Macmillan and Co., 1961.

strategy of the "bootstrap" operation and finally define the language in which new compilers are defined to the bootstrap.

A pictorial representation of the syntax of the new language for translation description and a sample "program" in that language (describing the translation of a conventional algebraic language) are supplied as appendices.

Introduction

The Translater is a general-purpose program which accepts strings in a source language, syntactically analyzes them, and emits a sequence of messages (which we call "macros"); the selection and emission of these messages may be triggered either by very local syntactic recognition or by rather complex investigation of syntactic context.

The emission of a macro actually consists of two activities: the construction of a macro descriptor and the call of a separate program (called the In-Sequence-Optimizer, or "ISO") which accepts and processes the descriptor.

The Translater is general-purpose in the sense that it is capable of syntactic analysis of any of a wide class of languages and of implementing any of a wide class of decision procedures for controlling macro emission by investigation of syntactic context. This generality is achieved by the use of tables which define the syntax of the source language and tables which define the decision procedure. The Translater itself is a relatively straightforward program which simply accepts a source language message and uses the tables to process it.

The "syntax tables" may be viewed as a coded representation of some standard formal definition of a language's syntax (for example, the Backus notation). Similarly, the second set of tables -- which we will call "generation strategy tables" -- may be viewed as an encodement of a class of statements defining the decision procedure. It is well known that a common formalism may serve for the definition of any of the usual programming languages and thus the possibility of using a standard program, with different tables for different languages, is clear. However, the existence of a convenient formalism for describing generation strategy -- and thus the possibility of a general-purpose algorithm controlled by a tabular encodement of a strategy -- is by no means as obvious.

We have devised such a formalism, which is adequate to describe fairly complex strategy more or less concisely.

The nature of the formalism in turn implies a table design and an associated algorithm -- the "generator" -- controlled by the tables.

The essential trick in the development of this formalism is the use of a tree-representation of the source-language message (which is the "natural" output of an analyzer). This tree-representation provides a convenient domain over which to define both contextual constraint and macro emission.

The formalism for describing generation strategy is itself a language -- "generation strategy language" -- and, in particular, a language describable in the Backus notation. This suggests the interesting possibility of a "bootstrap" compilation. That is, if one were to construct by hand:

1. A set of tables which define the syntactic structure of (say) the Backus notation (BNF) and the generation strategy language (GSL);
2. A set of tables which define the generation strategy for inspecting tree-representations of BNF and GSL formulas and emitting macros;
3. An algorithm which accepts these macros and interprets them as orders to build lines of a set of translater control tables;

Then one might, by putting this latter program in place of the ISO, use the compiler to create instances of the tables which control it.

In fact this is precisely what has been done. A program called the "Bootstrap ISO" creates, by processing macros, a set of prototypes of the control tables; then, after the entire BNF-GSL message has been processed an additional set of algorithms converts these prototype tables to final form. Since the actions performed by the Translater in executing the bootstrap translation are a proper subset of those it is capable of performing, and since the bootstrap runs better if the Translater can do a few things in a slightly different way, the Translater is "tuned" (slightly altered) in the bootstrap version; however, it is essentially the same, standard universal algorithm.

The remainder of this report is divided into two sections. Section A describes how the Translater works; Section B defines the bootstrap Translater by sketching the function of the Bootstrap ISO and defining the formalism BNFGSL -- the language which the bootstrap Translater in fact translates.

The Storage of Syntax Rules

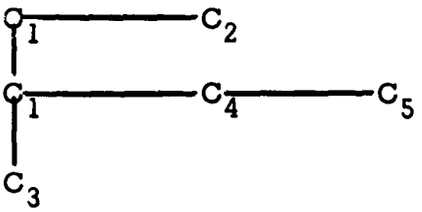
The formal languages in which we are interested may all be described (to within some special trickery for certain "small" types, like character strings) in the metalinguistics of the ALGOL 60 report* which has been called "b.n.f." (for Backus normal form). That is, each type is defined by a statement consisting of the name of the type enclosed by the signs "<" and ">" followed by the sign "::=" followed by a description of the alternative rules for forming the type, separated from each other by the sign "|". Each of these rules is in turn formed by concatenating "components" in the order in which they must be found to satisfy the rule. A component is given either by enclosing a type name in the signs "<" and ">" or by explicitly giving the exact symbols of the source alphabet to be recognized. Naturally, the metalinguistic signs are all excluded from the source alphabet.

Thus, each type may for our purposes be viewed as given by a form like:

$$\langle \text{type} \rangle ::= C_1\ C_2 \mid C_1\ C_4\ C_5 \mid C_3$$

where the C_1 represent either types or specific strings of characters.

The right-hand side of such a statement may be represented by a binary tree in which nodes correspond to components. The right son of a node is the next component in the rule and the left son is the first component of the next alternative rule. Thus, the example would be represented by:

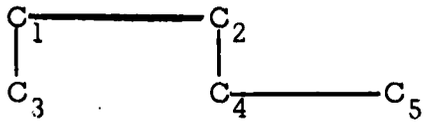


(We will conventionally draw right sons to the right and left sons below, in order to simplify layout.)

---

*"Report on the Algorithmic Language ALGOL 60", Naur(ed.) et al, Communications of the Association for Computing Machinery, Vol. 3, No. 5, May, 1960

This naturally suggests the application of the distributive law to shorten the statement and thus the tree. The right-hand side of the example means, conceptually, $C_1 \ (C_2 \mid C_4 \ C_5) \mid C_3$ and the corresponding tree

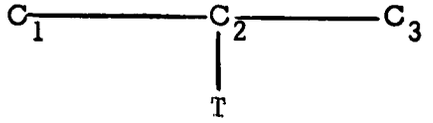


is what we clearly wish the analyzer program to work with. After all, the successful recognition of a C_1 followed by a failure to recognize a C_2 should lead directly to an attempt to recognize a C_4 instead of a C_2, rather than a re-investigation of alternatives to the entire formation.

It sometimes occurs that one formation for a type looks precisely like another followed by some additional component(s). Thus, we introduce a special symbol into the tree, which indicates successful completion of a formation even though a failure to continue it was encountered. This mark, which we will designate by "T" and call the "terminal" mark, is found only as a left son and may not have sons itself.

Example:

$C_1 \ C_2 \ C_3 \mid C_1$



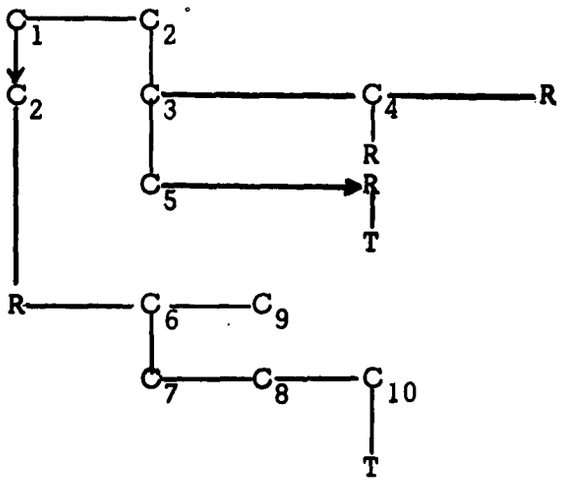
[Remark: It appears to be true of the source languages with which we are dealing that one is always safe looking for the longest formation for a type first and this assumption is made in order to shorten and simplify the analyzer algorithm.]

Recursive Definitions

If the definition of a type R includes a formation whose first component is R, we call that formation "left recursive"; we define "right

recursive" analogously. We specifically exclude any formation which mentions R twice, although we permit an arbitrary number of formations of R to mention R once. Left and right recursive definitions are not stored as written, but are revised to eliminate explicit mention of R.

Let us assume that a definition has been appropriately reordered so that it may be expressed as, for example:



The essential features of this reordering are as follows:

1. There is only one node N_o, representing R on the left; N_o has no left son.
2. The tree is really two distinct trees connected through N_1.
3. All nodes representing R (except for N_o) are in the upper tree with right sons of zero.

We process this tree as follows:

1. Take the right son of N_o (denoted N_1).
2. Erase N_o (setting left son of its father to zero).
3. Take every node representing R; replace it by a "null type" and set its right son to the trunk of the tree.
4. Take every node whose right son is zero and set its right son to N_1.

5. Take every node whose left son is T and set its left son to N_1.

6. Beginning with N_1, take left sons until a node N_2 is encountered with left son of zero. Set left son of N_2 to T.

The effect of all this processing is to command a re-entry of the trunk for all right recursives until a successful formation has been effected. Then, using the left recursive formations, efforts are made to extend it as long as possible.

A.2.

The Storage of Generation Strategy

Generation strategy is stated in terms of the nodes of a tree representation of a source language formula, and is composed as follows:

1. A set of rules for distinguishing kinds of node is given. Nodes are naturally distinguished by the syntactic types they represent; they may be further distinguished by their syntactic context.
2. Each distinct kind of node is assigned a short sequence of commands. Each command either defines an action to be performed (like, for example, the emission of a macro) or names another node of the tree.

The generator algorithm starts at the trunk of the tree and walks from node to node, at each node pausing to execute a command assigned to that kind of node. If the command indicates an action, that action is performed; if it names another node, the generator moves to that node. Upon each successive consideration of a node, the next command in that node's sequence of commands is considered. Whenever a considered node has no commands left, control moves automatically to the node's father in the tree.

Some actions have variables of call, which may be (among other things) the names of nodes of the tree. These names and the names of nodes given as commands and the names of nodes given in the context descriptions are all expressed by "relative tree names" -- that is, by rules for getting from any given point of the tree to some other. These rules take the form of a sequence of steps. Each step is either the name of a "neighbor" (e.g., FATHER, SON 3) or of some line item of a node. Typical relative tree names might be:

FATHER * RTSIB * SON 2

SON 2 * SON 3 * SON 1

At any given moment a relative tree name is interpreted with respect to the node which the generator is currently considering.

A.3.

Organization of the Translater Program

The Translater is controlled (after initialization) by a syntactic analyzer called ANALYZ.

ANALYZ is entered with a syntactic "goal": to recognize some specific syntactic type, starting at a specified character of the input string. Initially, the goal is "program", or its equivalent, and the starting character is the first. Every goal leads either to a new goal (if the current type is defined in terms of another type) or to an inspection of the input to find either a specific symbol string (stored in SSLIST), one of a finite set of specific symbol strings (stored in SSLIST), or one of a class recognized by a special scanner (e.g., an identifier or an integer). When a goal is found to be impossible to meet, the analyzer consults the syntax tables SYNTRE and SYNPT and either sets up an alternative goal at the same point of the input string or, if there are none, backs up the string to try an alternative interpretation of previous material.

Whenever a syntactic type which has been marked as "big" enough to call the generator (see B. 5.) is completely recognized, a call on the generator code, GENRAT, is made. During the course of recognition, a tree-representation (OUTREE) of the structure of that syntactic type was constructed.

GENRAT starts at the node (OUTREE line) which represents the syntactic type which caused the GENRAT call. At each node which it considers, GENRAT calls PREDIC to decide what kind of node it is.

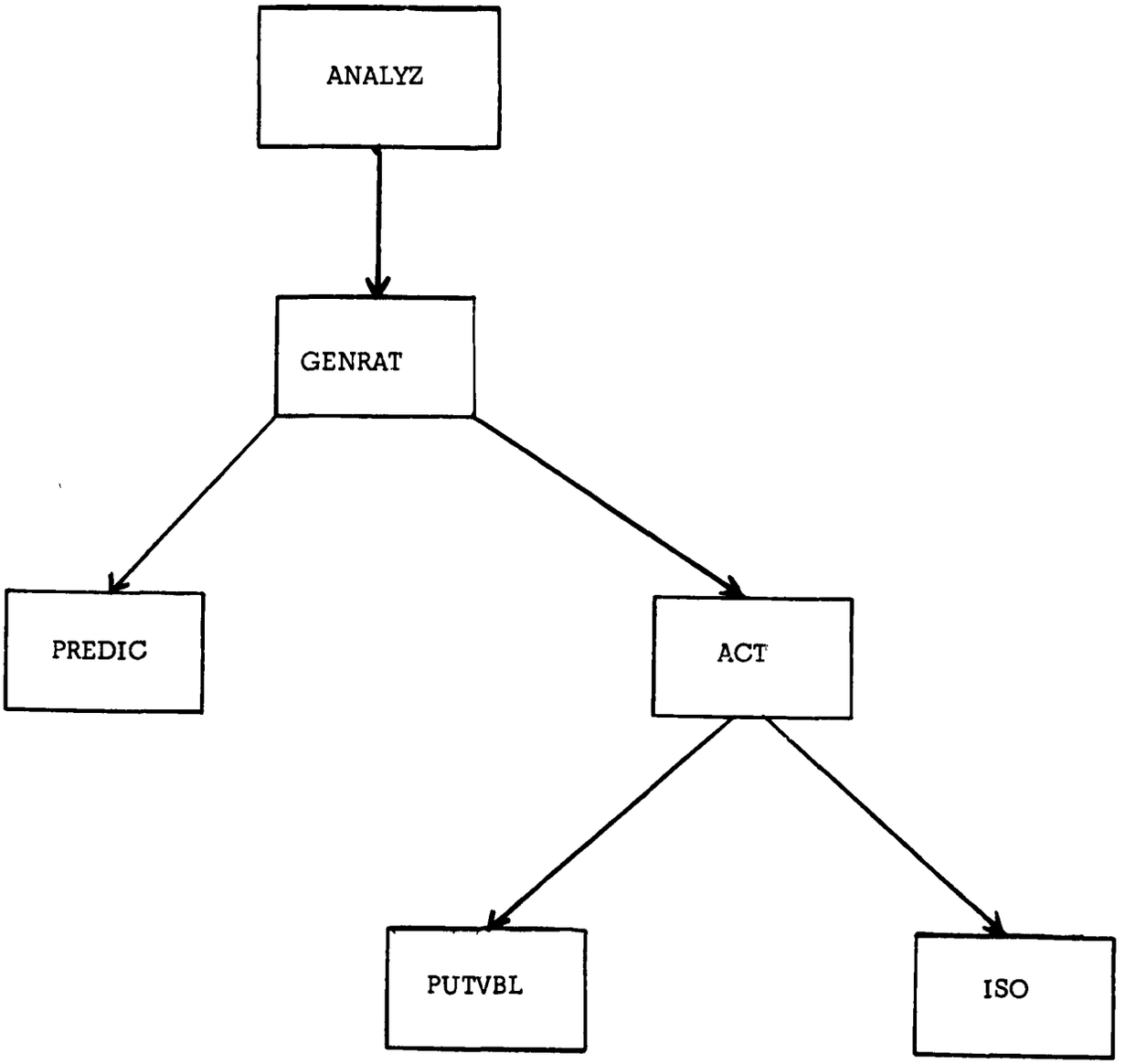
The algorithm PREDIC, using two control tables, considers both the syntactic type of the node and the nature of its context. The first of these control tables, PREDPT, contains an entry of each different kind of node. All kinds which are cases of the same syntactic type are stored contiguously in PREDPT and are pointed to by the SYNPT line for that type. Each line of PREDPT points in turn to the second control table, PRDCAT, in which is stored a list of (contextual) predicates to be satisfied before the node is adjudged to be of that kind. At the end of each list of predicates in PRDCAT is a pointer to the list of activities for nodes of that kind (see below).

Given PREDIC's result, GENRAT proceeds to do whatever need be done at nodes of that type on the n-th occasion they are considered. (GENRAT control keeps track of the current value of n for each node).

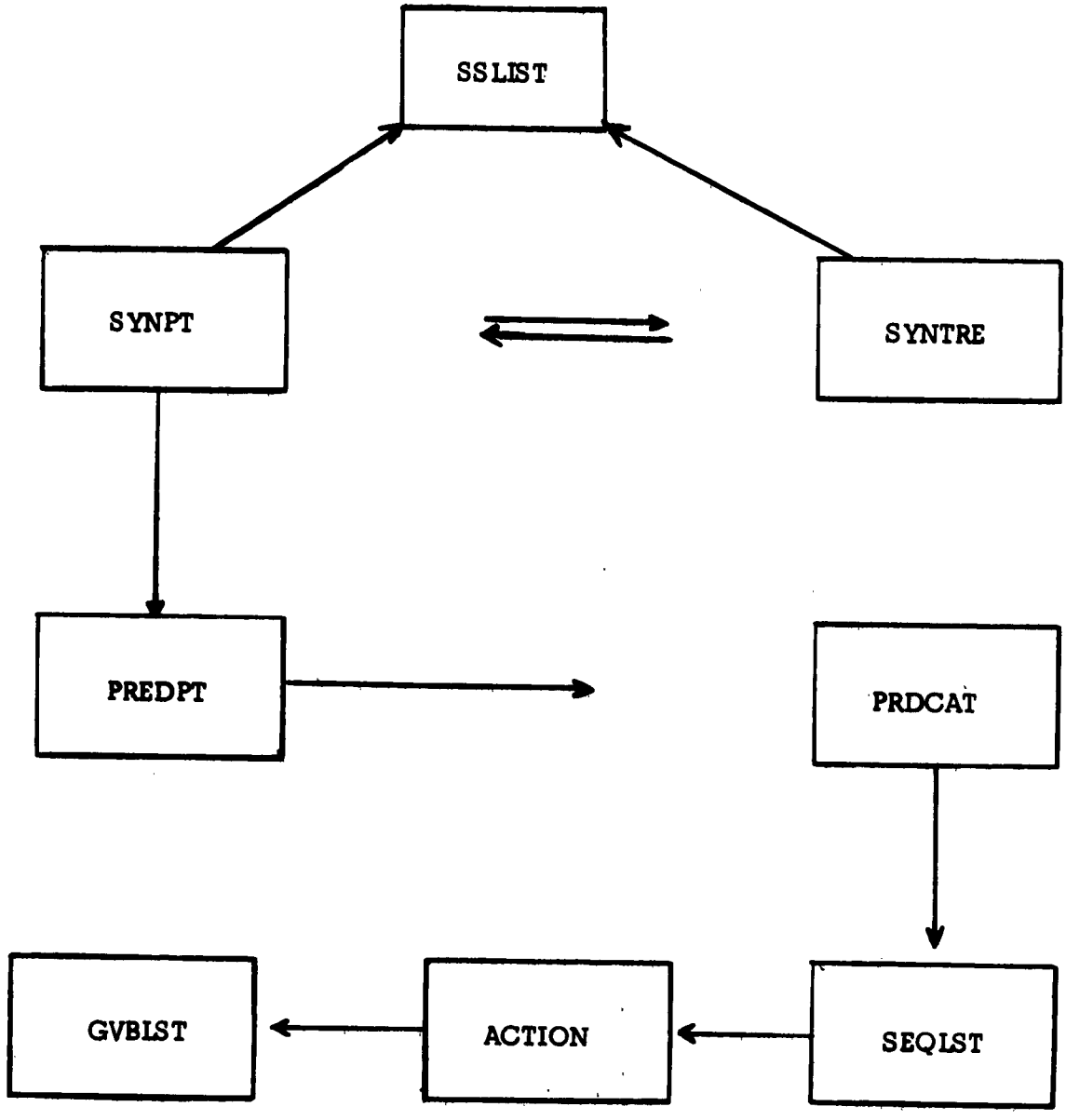
Lists of activities associated with different kinds of nodes are found in the table SEQLST. These activities either consist of transfer of consideration to another node or of a set of actions to be performed. If the latter, the set of actions is listed in the table ACTION. If any action has variables of call, these are listed in the table GVBLST. The routine ACT controls the execution of required actions.

Some actions involve the transmission of information to the optimizer. This is generally accomplished by putting the information into a communication table called SHPLST and calling the optimizer (ISO). Some of this information requires a certain amount of interpretation before being placed on SHPLST. The routine PUTVBL interprets certain kinds of encoded variable descriptions (found in GVBLST) and places its result in SHPLST.

Organization of the Routines



Organization of the Control Tables



B.

The Language BNFGSL

The language BNFGSL is a vehicle for defining to a "universal" compiler the particular translation it is to effect. A translater definition written in BNFGSL falls naturally into two main sections: first, a formal description of the source language, and second, a description of the "generation strategy" for getting from recognized constructs in the source language to corresponding computational commands.

The source language description controls a syntactic analysis whose output is a tree representation of the recognized structure (called OUTREE). When a "big" enough structure has been recognized, a call upon a generator is made. The generation strategy description is written in terms of a walk through this tree and has, as its central notion, the idea of a "relative tree name", by which we mean a rule for getting from the current node of the tree to some other node.

B.1.

The Definition of a Source Language

A language description is composed of a set of "type definitions" each of which defines a single syntactic type of the source language. In general, a syntactic type is defined by a set of "formations", or ways to form a representative of that type from smaller structures. There may be several alternative ways to form a type and thus a type definition may include several formations. Each formation is made up of a sequence of "components" which are to be concatenated to satisfy the formation. A component is either the name of some syntactic type or a fixed symbol string.

In BNFGSL, we use arbitrary identifiers to denote the names of syntactic types. Fixed symbol strings are written by the following rule: First write the character "$". Then write the characters of the desired symbol string, unless one of them is either "$" or "/" or a blank. In the latter case, precede each such character by a "$". The entire string is delimited by blank or "/". We begin a type definition by writing first the name of the type being defined, then the character "=", then each of the components of the first formation in order, then the character "/", then each component of the second formation, then another "/", and so on until there are no more formations; then we write "//".

Examples:

MULOP =$*/$$///

AREX = TERM /AREX ADOP TERM //

In addition to the set of formations for the type, we require that certain additional information be given as part of the type definition. This additional material is given in the form of a list of "tags", separated by commas, following the "//", and delimited by "/". The list of allowable tags and their meaning is as follows:

FIXLST - This type is defined by formations each of which has one component only, and that component is a fixed symbol string.

TLUAn (where n is an integer) - This type is defined, not by explicit formations, but by a special scanner -- specifically, the n-th such scanner.

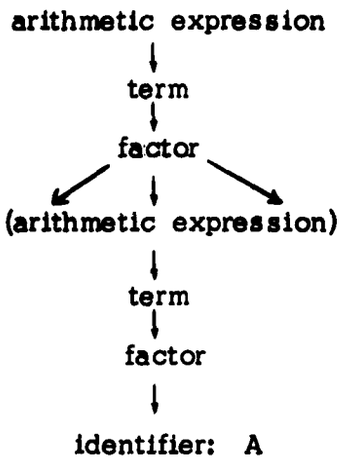
GENRAT - This type is big enough syntactically to warrant a call upon the generator.

INHIBT - This type is such that, while the analyzer is endeavoring to recognize it completely, no generation may occur (even if there is complete recognition of types which normally cause generation).

METAGN - This type is such as to trigger a call upon the generator; however, the purpose of that call is to process a declaration rather than to produce any code.

ASSIGN - This type is such that once it has been completely processed, any PRTECT's effected during its processing may be disregarded (see discussion of PRTECT, below).

INVOKE - Frequently, in the course of syntactic recognition, a substantial tree structure will be built in OUTREE, composed of a great many nodes of purely formal interest buried among which is a single interesting identifier. For example the simple construct "(A)" might lead -- through the syntax of many familiar languages --- to the structure:



Later on, during generation, it would be convenient to minimize investigation of this deceptively complex tree. The analyzer can detect these situations -- if given a few hints -- by carrying some of the properties of the components of each formation up to the node of the type being formed. The requisite hint is that those types which represent the variables of discourse (identifiers, integers, and strings, for example) or unary operators (addition operators, for example, if they may be unary) be tagged with "INVOKE". This will cause the automatic marking (during analysis) of each node as either "HARD" or "EASY" according as the subtree trunked by that node does or does not include at least two nodes of "INVOKE" type. Loosely speaking, "INVOKE" types are such that exactly those structures containing more than one of them require some code generations.

NULTYP - A "null type" is a type with no formations and is always successfully recognized without using up any characters.

The "language definition" section of a translater definition consists of a set of type definitions bracketed by the strings "BNF.." and "/BNF", respectively

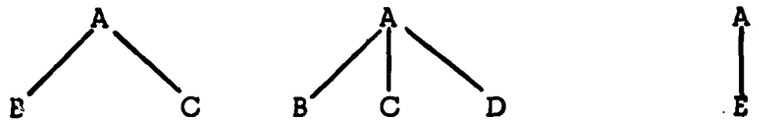
B.2.

The Structure of OUTREE

That structure in OUTREE which corresponds to any type recognition may be deduced from its formations. Thus if a type is given by:

A = B C / B C D / E ///

and an A is in fact recognized, the resulting structure would be one of the following:



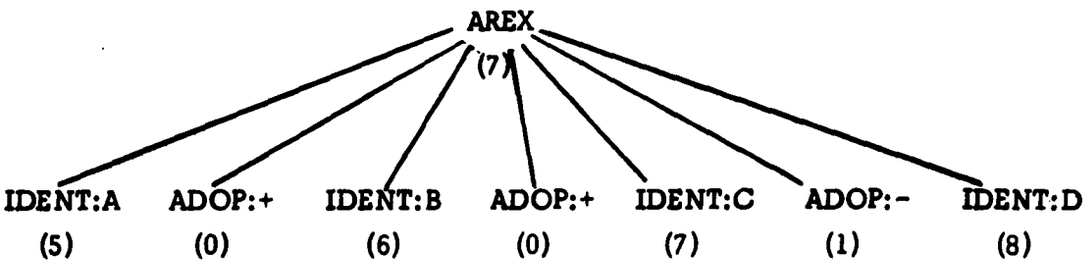
That is to say, the sons of a node correspond, 1 - 1 and in order, to the components of one of its formations, with two exceptions.

1. If a type is marked FIXLST, nodes which correspond to it will be terminal (have no sons).
2. If a formation is either left recursive or right recursive, it will result in a "bush" of components at the same level in the tree, with no explicit mention of the recursive type.

Example:

ADOP = $ + / $ - // FIXLST /
AREX = IDENT/AREX ADOP IDENT///

leads to the following representation of "A + B + C - D":



We refer to the nodes for the set of components of a type as the "sons" of the node for that type, numbered from left to right. We call the $(n+1)^{st}$ son, the "right sibling" of the nth. The meaning of "father" and "left sibling" is the obvious one.

The OUTREE actually contains a good deal of information at each node in addition to a code for the syntactic type represented. Most of this information is set and used by the compiler for its own internal purposes. Some items however are either used or set (or both) as a result of explicit indicators in the BNFGSL; these will be discussed below. For the moment, we merely remark that one item of a node, called "NR", always contains the appropriate one of the following:

1. The number of sons, if the node is not terminal.
2. The line number in the symbol table, the literal table, or the table of fixed symbol strings, corresponding to the node.
3. The relative number (starting with 0) of the formation, if the node represents a fixed list (FIXLST type).

The parenthesized figures in the previously given tree are NR values, assuming that "A", "B", "C", "D" are in lines 5, 6, 7, 8 of the symbol table, respectively. Neither the identifiers themselves nor the BCD-representations of "+" and "-" are carried explicitly in the nodes.

B.3.

The Definition of a Generation Strategy

The "generator definition" section of a translater description is composed of a set of "generation statements" bracketed by "GSL.." and "/GSL", respectively.

Each generation statement begins with a "condition" to be satisfied; then there follows (after a delimiting ",") a set of activities to be performed if the condition is satisfied. A condition begins with "IF" followed by either the name of a syntactic type or a fixed symbol string. Then there may be an arbitrary number of additional "predicates" to be satisfied. The different predicates are concatenated with "AND"s; that is, they must all be satisfied in order for the condition to be met.

There are six kinds of predicates, which are as follows (where, by "<R>" we mean any representative of the type R and in particular, by "<r.t.n.>" we mean any relative tree name):

	Type	Form	Meaning
1.	Zero test	<r.t.n.> = 0	True, if the named node is on the tree.
		<r.t.n.> X = 0	Converse
2.	Syntactic type test	<r.t.n.> IS <identifier>	True, if the syntactic type of the named node is as indicated.
3.	Symbol test	<r.t.n.> = <fixed symbol string>	True, if the named node is precisely the string indicated; it is assumed that the node will represent either a fixed symbol string or some syntactic type tagged with "FIXLST" or an identifier.

	Type	Form	Meaning
4.	Complexity test	<r.t.n.> IS HARD <r.t.n.> IS EASY	Tests the named node for the property described in the section on the tag "INVOKE", above.
5.	NR test	NR(<r.t.n.>) = <integer>	True if the NR field of the named node has the indicated value.
6.	VTYP test	VTYP(<r.t.n.>) = <integer>	It would be convenient if declared or encountered properties of the indentifiers could be discussed in BNFGSL. In particular, it is essential to be able to consider the "variety" of the identifier (function, packed item name, or whatever), which is presumed filled into the VARITY item of the symbol table by the declaration processors. This test, which is well-defined only if the named node designates an identifier, is true if the VARITY field of the SYMTAB line number given in the NR field of the named node has the required value.

Examples:

IF TERM AND SELF IS HARD,

IF $(AND VTYP(FATHER*LFTSIB) = 6,

IF VARBLE AND SON2 X = 0 AND FATHER IS FACTOR,

If a given node satisfies several different conditions, it will be treated as though it satisfied the first in order of appearance in the BNFGSL.

Following the condition and the delimiting "," is a "sequencing list" composed of an arbitrary number of "sequencing elements" each preceded by the character "$". The entire sequencing list is delimited by a ".".

A sequencing element is either a single relative tree name or an "action list", by which we mean a set of "actions" to be performed. Each action is given in functional notation; i.e., each is composed of an "imperative" naming the action followed by a list of its variables of call separated by commas and bracketed by a pair of parentheses. Actions with no variables of call are given simply as imperatives.

In order to understand the motivation of this section of the language, it is helpful to view the generator as at any time considering some particular node of OUTREE (initially that which triggered the GENRAT call). It is this node with respect to which relative tree names are evaluated. A node may be considered several times. When the generator considers a node for the first time, it is determined whether that node satisfies one of the conditions given in the BNFGSL. If it does, that node is set up to perform the activities indicated in the sequencing list associated with that condition. The first time that node is considered, it will perform whatever is indicated by the first sequencing element; the second time, the second element; and so on. If an element is simply a relative tree name, the generator will go on to consider the node named by that element. Otherwise, the generator will cause the execution of all the actions on the action list which comprises that element.

If a sequencing element is a relative tree name and if that name is not well-defined with respect to the node currently being considered, the generator moves to the next sequencing element.

If we have run out of sequencing elements for a node or if a node never meets any condition in the first place, subsequent attempts to consider

that node will always lead to immediate consideration of its father. If that node is the node which triggered generation, the automatic walk to its father is the signal which causes return of control to the analyzer.

The imperatives are as follows:

OUTPUT - This has a variable number of arguments, but at least one; its effect is to throw out onto a communication table (SHPLST) all the arguments in order and then execute a call upon the ISO. The first argument position is conventionally used to transmit the name of the macro-instruction being output. Thus, OUTPUT(SWITCH,SON3,NR(SON5)).

PUTVLA -
PUTVLB -
OUTPOT - PUTVLA places its single argument in the first line of the communication table SHPLST. PUTVLB places its single argument in the next free line of SHPLST. OUTPOT acts like OUTPUT, except with no arguments. Both PUTVLA and PUTVLB interpret their arguments as values (see discussion of MINE, below).

PUTNAM - This places the OUTREE line number of its single argument on the next free line of SHPLST.

LABEL - This causes the current point in the output to be labeled with the identifier given by its single argument. Example:

LABEL(SON1)

MINE - Several imperatives (notably OUTPUT, PUTVLA, and PUTVLB) give as arguments relative tree names whose intended interpretation is as values. If a relative tree name leads to a terminal node denoting an identifier, the interpretation is direct. If, however, it leads to a non-terminal node, this may mean a deferred reference to another terminal node (if the non-terminal node is EASY) or may refer to the result of a previously output computation. This means that a certain amount of interpretation must be done automatically by the generator. In order to provide

enough information to enable this interpretation, the generator follows all executions of OUTPUT or OUTPOT by marking the node whose consideration directed the call with a code number for that line of output. If later some relative tree name leads to that node, the interpretive machinery will deduce (by inspecting HARD-EASY codes, etc.) that this node should have led to output and then obtain the output line number from it.

A difficulty arises when the node N with which the output is to be associated for later interpretation is different from the code N^1 whose consideration led to the output call. If the tree structure is simply such that N^1 is the trunk of the only interesting subtree of the tree trunked by N, (by only "interesting", we mean that nothing outside it contains any INVOKE type codes), then the generator can quickly locate the interesting N^1 and proceed. If this situation does not obtain, however, some hint must be given the generator. The action MINE has the effect of associating, with the node currently under consideration, the last line which was output.

Example:

```
IF AREX AND SELF IS HARD,
    $ SON1 $MINE.
```

The result of this would be that, no matter which node in the subtree trunked by the arex actually output the last line in the computation of its value, future interpretation of the arex node as a value would correctly be set to the last line of output.

SET
COPY

Two fields are associated with each node, which may be used to transmit OUTREE line numbers from node to

node. They are called "TRUE" and "FALSE", respectively, since they are used commonly to pass around, within a boolean expression, the true and false jump destinations. The ordered pair composed of these two fields is called "TRUTAB".

SET and COPY each has two arguments, of which the first is either "TRUTH" or "FALSTY" for both and "TRUTAB" for COPY only. The second argument is, in all cases, a relative tree name. For SET, the meaning is to set the value of the indicated field (TRUE for TRUTH, FALSE for FALSTY) of the node currently being considered to the OUTREE line number obtained by evaluating the relative tree name). For COPY, the meaning is to copy the designated field in the named node into the corresponding field of the node currently being considered.

Examples:

```
SET(TRUTH, RTSIB*RTSIB)
COPY(TRUTAB, FATHER)
```

BEGRGN
ENDRGN

These bracket a "region"; by which we mean a segment of code suitable in size and character as a domain for special register assignment optimization. These imperatives take no arguments.

BEGLUP
ENDLUP

These bracket a region which is also a loop. Their effect upon the ISO is the same as that of BEGRGN and ENDRGN, save that in addition a search is made for invariant parts of the computation so that they may be moved out of the loop. No arguments.

BEGRTN
ENDRTN } These bracket a "routine", or file-able entity. No arguments.

BEGCND This pair brackets a section of a boolean expression which may be jumped around (if booleans are handled by conditional transfers). Its purpose is to prevent the common subexpression detectors in the ISO from assuming that certain values which may be already computed, always are. No arguments.
Example: (The brackets indicate possible jumped-over regions):

IF A EQ B+C OR B EQ C+D AND Z EQ W+Y, ...

PRTECT - This imperative directs the ISO to prevent destruction of the current value of its single argument (which must be a relative tree name denoting an identifier) until that value is used. The purpose of this device is to protect against situations where embedded assignment statements alter the intent of an arithmetic expression or assignment. For example, consider:

A(I) = X + (I = I + 1)

If the intention is for A(I) to be interpreted in terms of the old value of "I", one can emit a PRTECT of the node representing the first "I" before emitting outputs which cause the computation of the righthand side of the assignment. The ISO will then cause the old "I" to be saved before it is over-written and reference the saved value when treating A(I).

EXECUT - This provides a mechanism for ordering the run-time execution of an arbitrary, independent, "hand-coded"

program. It has one argument, the name of the program. The Bootstrap assigns integers to the identifiers which appear as arguments of EXECUT in order of first appearance starting with one. At run time the EXECUT is interpreted as a jump through switch block to the n-th independent algorithm, where n was the integer assigned. It is possible to force the numbers assigned either by giving the desired integer explicitly as the argument to EXECUT or (better) by initializing the table (EXTPGM) in which such identifiers are looked up and (appended if not already there) by the Bootstrap. If it is desired to provide arguments to an independent algorithm, they can be placed on SHPIST by appropriate use of PUTVLA, PUTVLB, and PUTNAM.

B.4.

Relative Tree Names

A "relative tree name" is of one of four kinds: either the fixed string "SELF" (meaning the node currently under consideration), the fixed string "NXTREE" (meaning the trunk of the next tree generated over), a "tree walk", or a tree walk bracketed by "LAST(" and ")".

A tree walk is a sequence of "steps" (encoding requirements impose a practical limit on the number: say, seven), separated by "*". Steps are chosen from the set:

FATHER -	father
RTSIB -	right sibling
LFTSIB -	left sibling
SONLST -	rightmost (last) son
TRUE -	true field
FALSE -	false field
SON n -	n^{th} son, where $1 \leq n \geq 10$

Example:

SON1*SON2

FATHER * RTSIB*TRUE*RTSIB

The interpretation of such a name is that (starting at the node under consideration) one executes the steps in order, yielding as value an OUTREE line number. If at any point a step is ill-defined (walks off the tree), the value of the name is set to zero.

If the walk was bracketed by "LAST(" and ")" and also walked off the tree, then the value is set to the line number resulting from the last step which stayed on the tree.

B.5.

Sample Translater Description

As a brief example, let us define an extremely simple language translation in BNFGSL:

```
BNF..
IDENT = // TLUA1, INVOKE/
TERM = IDENT / TERM MULOP IDENT///
AREX = TERM / AREX ADOP TERM///
ADOP = $+ / $- // FIXLST/
MULOP = $* / $$ / // FIXLST/
ASGST = IDENT $ = AREX $. // GENRAT/
/ BNF
GSL..
IF ASGST, $ SON3 $ OUTPUT (STORE, SON1, SON3).
IF AREX AND SELF IS HARD, $ SON1 $ MINE.
IF TERM AND SELF IS HARD, $SON1 $MINE $LFTSIB $RTSIB*RTSIB.
IF TERM, $LFTSIB $RTSIB*RTSIB.
IF IDENT, $LFTSIB $RTSIB*RTSIB.
IF ADOP, $OUTPUT (ADDSUB,NR(SELF), LAST (LFTSIB*LFTSIB),
    RTSIB) $RTSIB*RTSIB*RTSIB.
IF MULOP, $ OUTPUT ( MPYDIV, NR(SELF), LAST (LFTSIB*LFTSIB),
    RTSIB) $RTSIB*RTSIB*RTSIB.
/ GSL
```

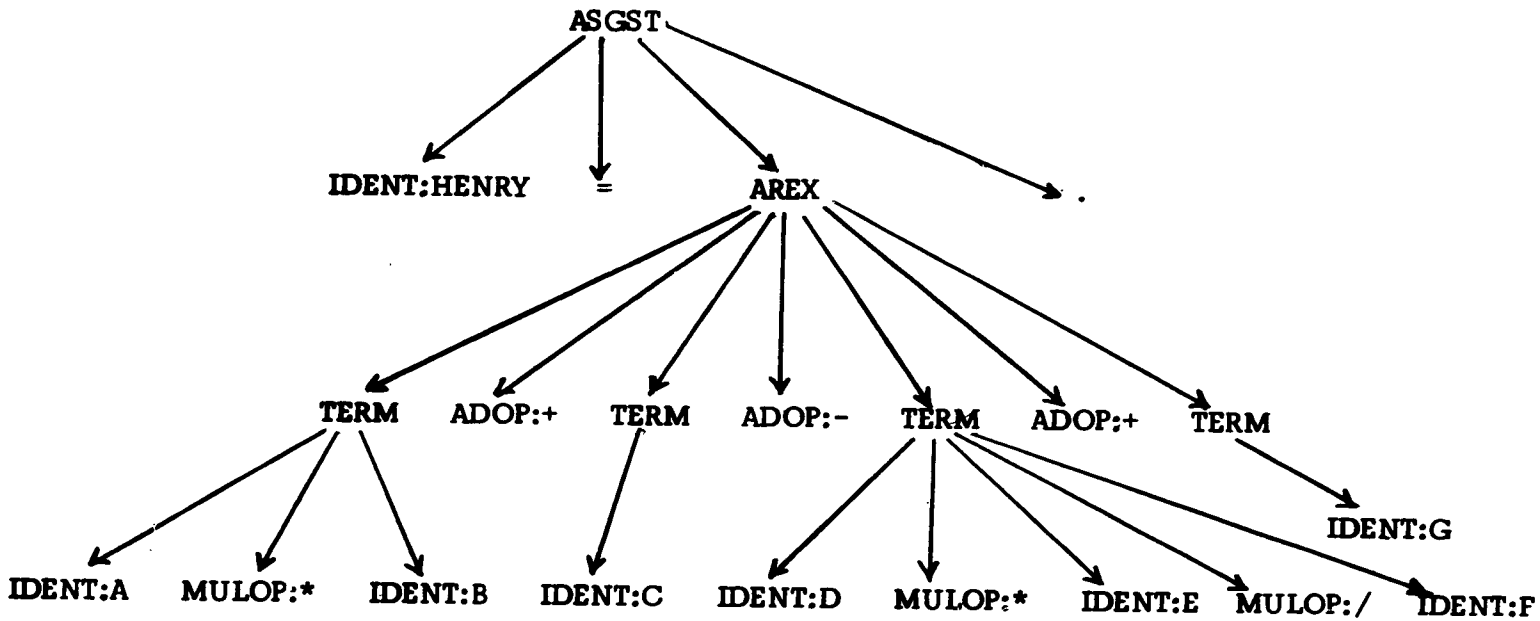
If the above deck is input to the Bootstrap, a set of tables for the Translater will be generated. When the Translater, with that set of tables and an appropriate identifier scanner, receives as input:

HENRY = A*B+C-D*E/F+G.

it will effectively output to the ISO the following:

1. MPYDIV, 0, A, B
2. ADDSUB, 0, line 1, C
3. MPYDIV, 0, D, E
4. MPYDIV, 1, line 3, F
5. ADDSUB, 1, line 2, line 4
6. ADDSUB, 0, line 5, G
7. STORE, HENRY, line 6

The OUTREE which is constructed by the analyzer will look as follows:


ASGST
IDENT:HENRY
=
AREX
.
TERM
ADOP:+
TERM
ADOP:-
TERM
ADOP:+
TERM
IDENT:G
IDENT:A
MULOP:*
IDENT:B
IDENT:C
IDENT:D
MULOP:*
IDENT:E
MULOP:/
IDENT:F

B.5.

Formal Description of BNFGSL

This section contains a description of the translation of BNFGSL written in BNFGSL.

The syntax-descriptive section of this translation description should be comprehensible by reference to the previous discussion of BNFGSL. However, it is difficult to understand the motivation for the macros output to the Bootstrap ISO without at least a sketchy idea of what the latter algorithm does with them.

In brief, each macro emitted to the Bootstrap ISO corresponds to a line of one of the seven prototype control tables being constructed. These seven tables are prototypes of the following; respectively:

1. ACTION
2. GVBLST
3. PRDCAT
4. PREDPT
5. SEQLST
6. SYNPT
7. SYNTRE

The literal symbol table of the bootstrap compilation becomes the eighth control table, SSLIST.

The first variable of call of the OUTPUT imperative names, not a macro, but the number (1 through 7) of that prototype table of which this call provides a line. The remaining variables of call correspond to the line items of the line and require, for their completer understanding, a discussion of the tables of the translater far too detailed for the scope of this report. It should be noted however (by reference to the picture of table organization in A.3.) that several tables contain line items which point to other tables. The seven line counters of the prototype tables being built supply the numbers of their "first-unfilled" lines to the Bootstrap ISO and thus this sort of item need never be given explicitly.

```
BNF..
SYNTYP = // TLUA1 /
INTGER = // TLUA2 /
SYMSTG = // TLUA3 /
LNGDEF = $BNF.. TYPLST $$/BNF //GENRAT /
TYPLST = TYPDEF/TYPLST  TYPDEF ///
TYPDEF = SYNTYP  $ = FSTPRT SECPRT //  GENRAT/
FSTPRT = FORMST $$/$/  /$$/$/  ///
SECPRT = TAGLST $$/  /$$/  ///
TAGLST = TAG/TAGLST $, TAG///
TAG = $INVOKE/$ASSIGN/$GENRAT/$METAGN/
    $INHIBT/$FIXLST/$TLUA  INTGER/$NULTYP///
FORMST = FORM/FORMST $$/  FORM///
FORM = COMP / FORM  COMP ///
COMP = SYNTYP/SYMSTG ///
IDENT = //  TLUA1 /
GENDEF = $GSL.. GNSLST  $$/GSL///
GNSLST = GENSTA/GNSLST  GENSTA ///
GENSTA = CNDITN $, SEQLST $. //  GENRAT/
CNDITN = KIND/KIND PRDLST ///
KIND = $IF SYNTYP/$IF SYMSTG ///
PRDLST = $AND PREDCT/PRDLST $AND PREDCT///
PREDCT = ZERTST/VTPTST/SYMTST/CPXTST/NRTST/SYNTST///
ZERTST = TRENAM EQREL ZERO///
ZERO = $0 ///
SYNTST = TRENAM COPULA SYNTYP///
SYMTST = TRENAM COPULA SYMSTG///
CPXTST = TRENAM $IS HRDNES ///
HRDNES = $EASY /$HARD// FIXLST/
NRTST = NRTRE $= INTGER ///
NRTRE = $NR(  TRENAM $)///
```

```
VTPTST = VTPTRE $= INTGER ///
VTPTRE = $VTYP (  TRENAM  $) ///
EQREL = $= /  $ x= //  FIXLST/
COPULA = $= / $IS //  FIXLST /
SEQLST = $$$ SEQUEL/  SEQLST $$$ SEQUEL///
SEQUEL = TRENAM / ACTLST ///
ACTLST = ACT /  ACTLST ACT ///
ACT = IMPER /  IMPER $( VELIST $) ///
IMPER = $OUTPUT/  $LABEL /$MINE/$SET/$COPY/
$PUTVLA/$PUTVLB/$EXECUT/$OUTPOT/$BEGRTN/
$ENDRTN/$BEGRGN/$ENDRGN/$BEGLUP/$ENDLUP/
$BEGCND/$ENDCND/$PUTNAM/$PRTECT// FIXLST/
VBLIST = VBL/VBLIST $, VBL ///
VBL = TRENAM / INTGER/ SETTEE / NRTRE/IDENT///
SETTEE = $TRUTH/$FALSTY/$TRUTAB//  FIXLST/
TRENAM = TREWLK /$LAST (  TREWLK  $)/
    $SELF /  $NXTREE///
TREWLK = STEP/ TREWLK $* STEP ///
STEP = $FATHER/$LFTSIB/$RTSIB/$SONLST/
    $TRUE/$FALSE/$SATLPT/$SON1/$SON2/
    $SON3/$SON4/$SON5/$SON6/$SON7/
    $SON8/$SON9/$SON10// FIXLST/
XLATER = LNGDEF GENDEF///
/ BNF
```

```
GSL..
IF  TYPDEF, $LABEL (SON1) PUTVLA(6)
    $SON4  $PUTVLB(30) OUTPOT  $SON3.
IF  SECPRT  AND  SON2 = 0, $ FATHER.
IF  SECPRT, $SON1*SON1.
IF  TAG  AND  SON2 = 0, $PUTVLB (NR(SON1))
    $RTSIB * RTSIB.
IF  TAG, $PUTVLB (NR(SON1))  PUTVLB (NR(SON2))
    $RTSIB * RTSIB.
IF  FSTPRT  AND  SON2 x=  0, $SON1 * SON1.
IF  FORM, $SON1 $RTSIB * RTSIB.
IF  COMP AND SON1  IS  SYNTYP  AND
    LFTSIB = 0, $ OUTPUT (7, 0, NR(SON1),
    FATHER * RTSIB * RTSIB, RTSIB)
    $RTSIB.
IF  COMP  AND  SON1  IS  SYNTYP,
    $OUTPUT (7, 0, NR(SON1), 0, RTSIB)
    $RTSIB.
IF  COMP  AND  LFTSIB = 0,
    $OUTPUT  (7, 1, NR(SON1), FATHER * RTSIB *
    RTSIB, RTSIB)  $RTSIB.
IF  COMP, $OUTPUT (7, 1, NR(SON1), 0, RTSIB)
    $RTSIB.
IF  GENSTA, $ SON1 $SON3.
IF  CNDITN, $SON1 $SON2 $OUTPUT (3, 0).
IF  KIND  AND  SON2 = SYNTYP,
    $OUTPUT (4, NR(SON2)).
IF  KIND, $OUTPUT (4, 0)  OUTPUT (3, 3, 0, NR(SON2)).
IF  PRDLST, $SON2.
IF  PREDCT  AND  SON1 = ZERTST,
    $ SON1 * SON1  $OUTPUT (3, 1, NR(SON1 * SON2))
    $RTSIB * RTSIB.
```

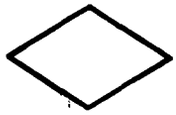
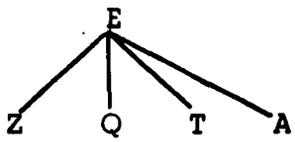
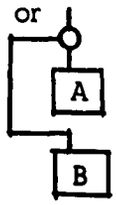
```
IF PREDCT  AND  SON1 = SYNTST, $SON1 *SON1
    $OUTPUT (3, 2, NR (SON1 * SON3))  $RTSIB * RTSIB.
IF PREDCT  AND  SON1 = SYMTST, $SON1 * SON1
    $OUTPUT (3, 3, NR(SON1 * SON3))  $RTSIB * RTSIB.
IF PREDCT  AND  SON1 = CPXTST, $SON1 * SON1
    $OUTPUT (3, 4, NR(SON1 * SON3))  $RTSIB * RTSIB.
IF PREDCT  AND  SON1 = NRTST, $SON1 * SON1 * SON2
    $OUTPUT (3, 5, NR (SON1 * SON3 ))  $RTSIB * RTSIB.
IF PREDCT, $SON1 * SON1 * SON2
    $OUTPUT (3, 6, NR (SON1 * SON3))  $RTSIB * RTSIB.
IF SEQLST, $SON2 $OUTPUT (5, 2).
IF SEQUEL  AND  SON1  IS  TRENAM,
    $SON1  $OUTPUT (5, 0)  $RTSIB * RTSIB.
IF SEQUEL, $OUTPUT (5, 1) $SON1  $RTSIB * RTSIB.
IF ACTLST, $ SON1  $OUTPUT (1, 30).
IF ACT, $OUTPUT (1, NR(SON1))  $SON3
    $RTSIB.
IF VBLLST, $SON1  $OUTPUT (2, 0).
IF VBL  AND  SON1 = INTGER,
    $OUTPUT (2, 1, NR(SON1))  $RTSIB * RTSIB.
IF VBL  AND  SON1 = TRENAM  AND
    LFTSIB * LFTSIB * SON1 = SETTEE, $LFTSIB*
    LFTSIB.
IF VBL  AND  SON1 = TRENAM, $SON1
    $OUTPUT (2, 2) $RTSIB * RTSIB.
IF VBL AND SON1 = NRTRE, $SON1 *SON2
    $OUTPUT (2, 4) $RTSIB * RTSIB.
IF VBL  AND  SON1 = IDENT, $OUTPUT (2, 3, NR (SON1))
    $RTSIB * RTSIB.
IF VBL  AND  SON1 = SETTEE, $RTSIB *
    RTSIB * SON1  $OUTPUT (2, 5, NR(SON1)).
IF TRENAM  AND  SON2 x= 0, $PUTVLA (1)
    $SON2  $EXECUT (FILTRE).
```

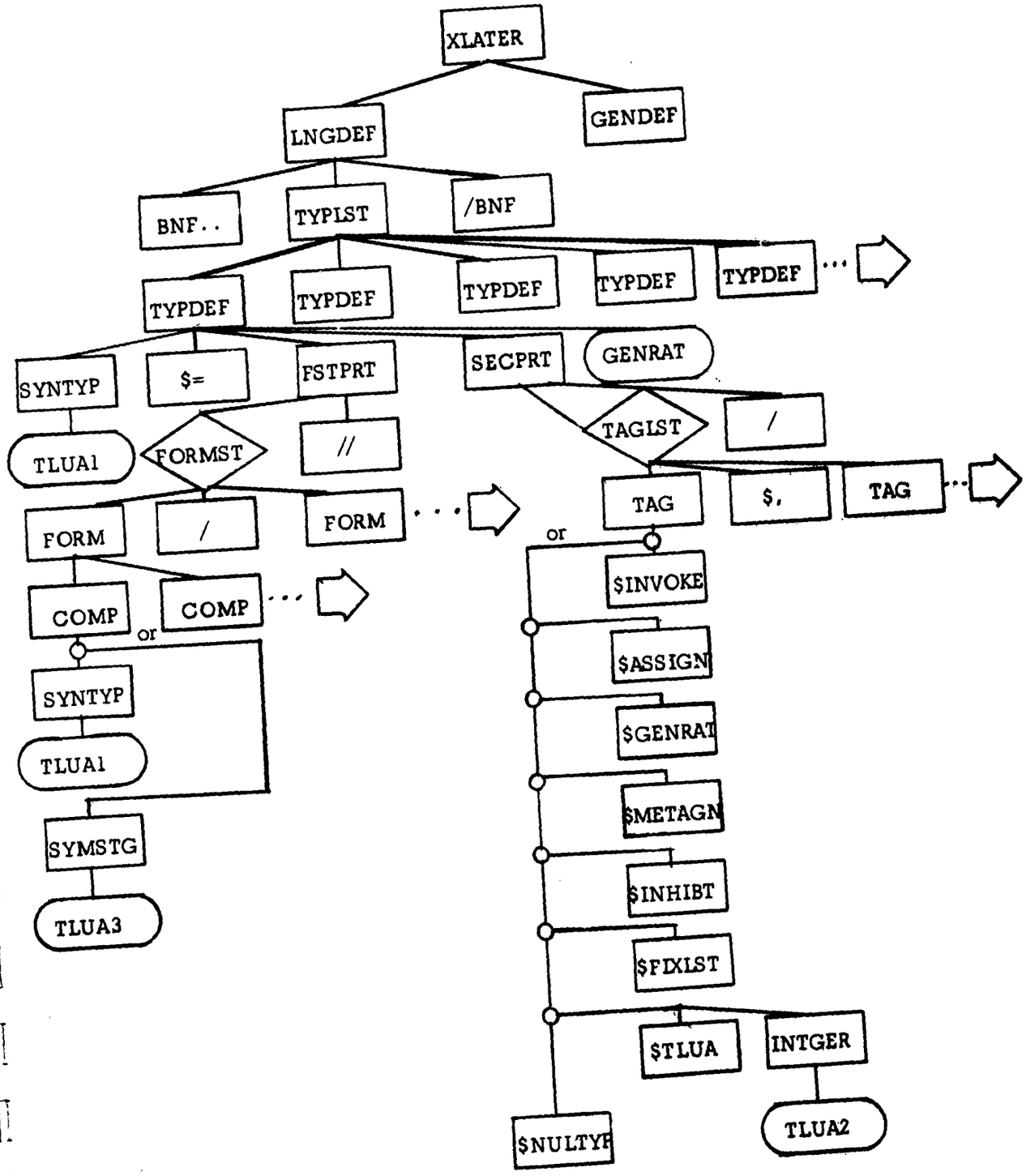
```
IF TRENAM    AND    SON1 = $NXTREE   , $PUTVLA (3)
    EXECUT (FILTRE).
IF TRENAM, $PUTVLA (0) $SON1   $EXECUT (FILTRE).
IF TREWLK, $SON1   $PUTVLB(30).
IF STEP, $PUTVLB (NR(SELF))   $RTSIB * RTSIB.
/ GSL
```

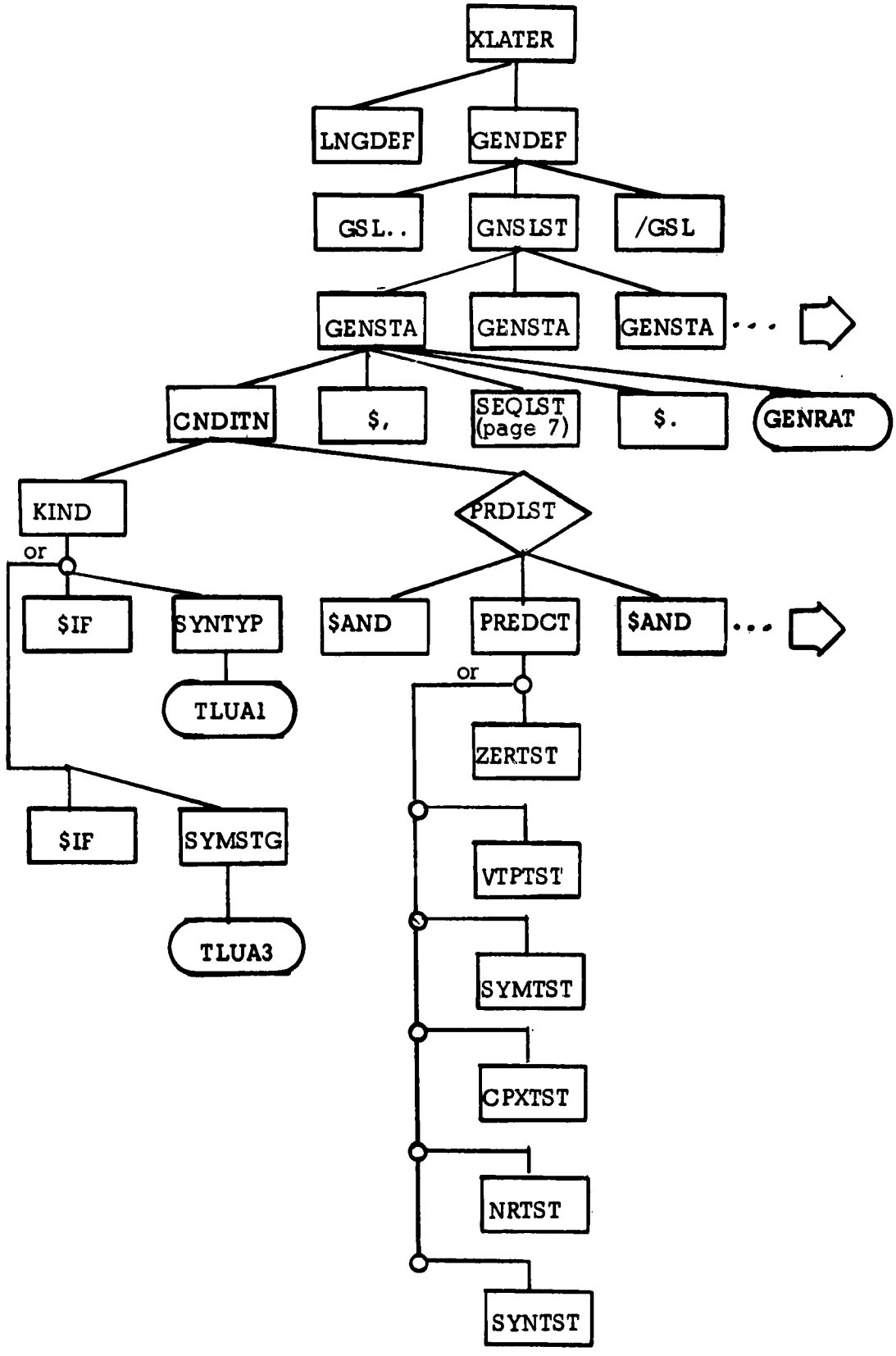
Appendix I

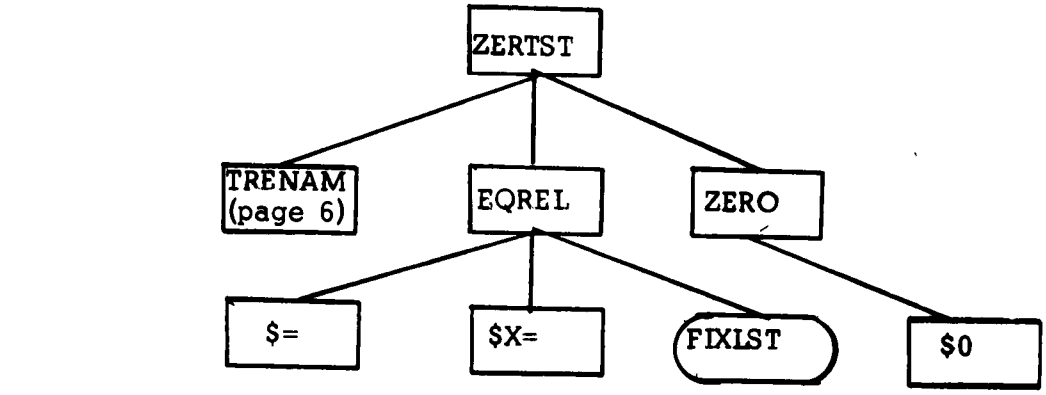
To assist the reader in visualizing the syntactic structure of BNFGSL, we give here a pictorial representation of that syntax:

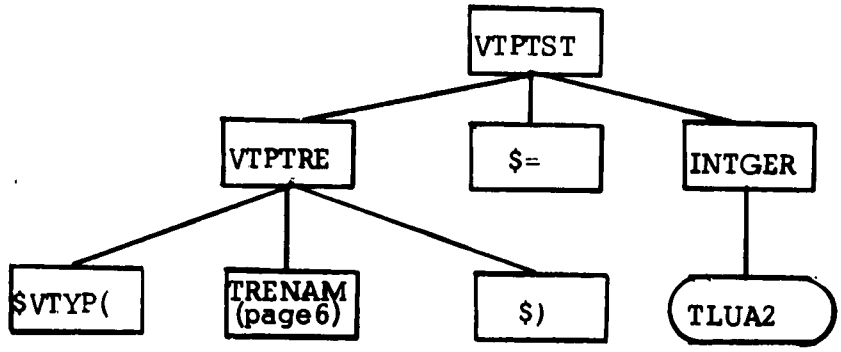
LEGEND

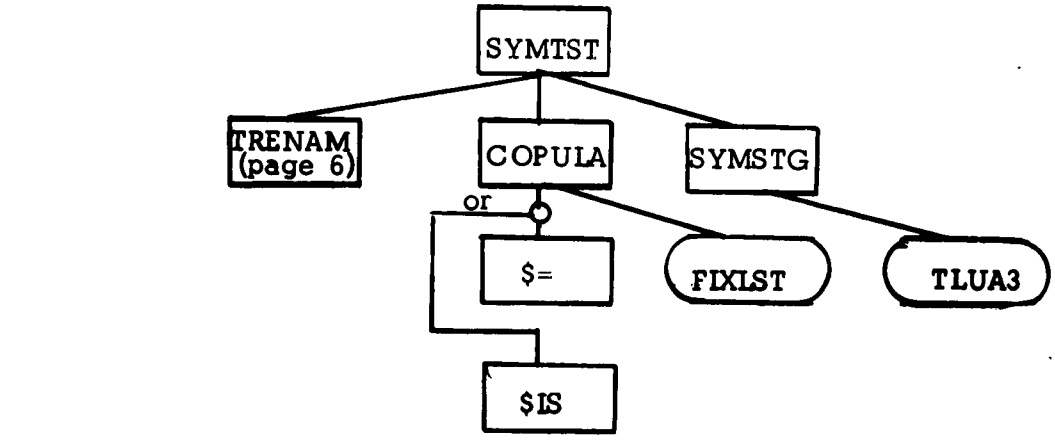
SYMBOL	DEFINITION
	Enclosed in this symbol is a syntactic type or a symbol string.
	This symbol contains the name of a tag involved with a syntactic type.
	This symbol contains an optional syntactic type or symbol string.
	Lines or rays emanating from a single point denote sequential components. Thus, Z Q T A are the sequential components of E.
	The circle denotes an OR junction. Here we have A or B.
	This symbol denotes the beginning of a symbol string.

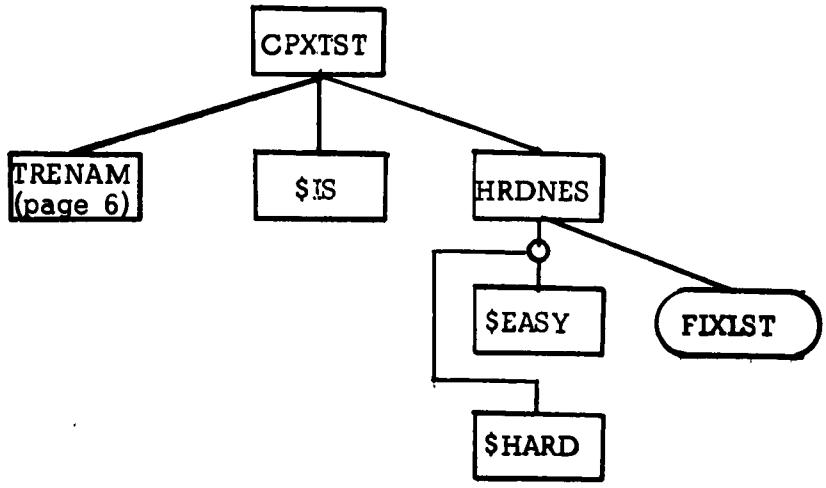

XLATER
LNGDEF
GENDEF
BNF..
TYPLST
/BNF
TYPDEF
TYPDEF
TYPDEF
TYPDEF
TYPDEF
SYNTYP
$=
FSTPRT
SECPRT
GENRAT
TLUA1
FORMST
//
TAGLIST
/
FORM
/
FORM
TAG
$.
TAG
COMP
COMP
or
SYNTYP
TLUA1
SYMSTG
TLUA3
$INVOKE
$ASSIGN
$GENRAT
$METAGN
$INHIBT
$FIXLST
$TLUA
INTGER
$NULTYP
TLUA2

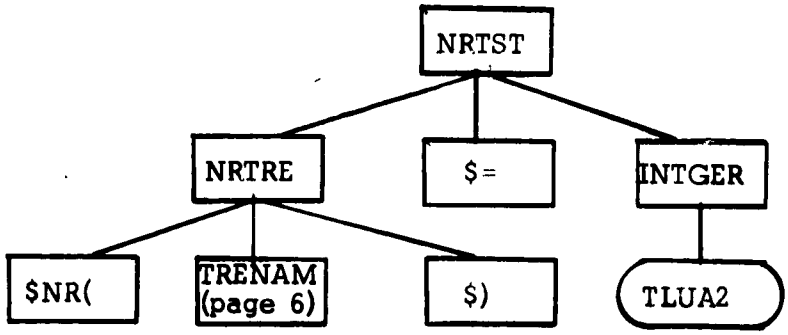

XLATER
LNGDEF
GENDEF
GSL..
GNSLST
/GSL
GENSTA
GENSTA
GENSTA
CNDITN
$,
SEQLST
(page 7)
$.
GENRAT
KIND
or
PRDLST
$IF
SYNTYP
$AND
PREDCT
$AND
TLUA1
or
ZERTST
$IF
SYMSTG
VTPTST
TLUA3
SYMTST
CPXTST
NRTST
SYNTST

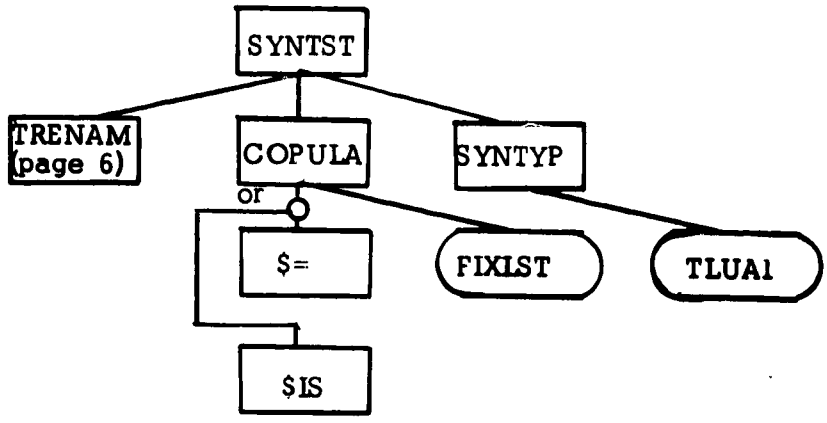

ZERTST
TRENAM (page 6)
EQREL
ZERO
$=
$X=
FIXLST
$0

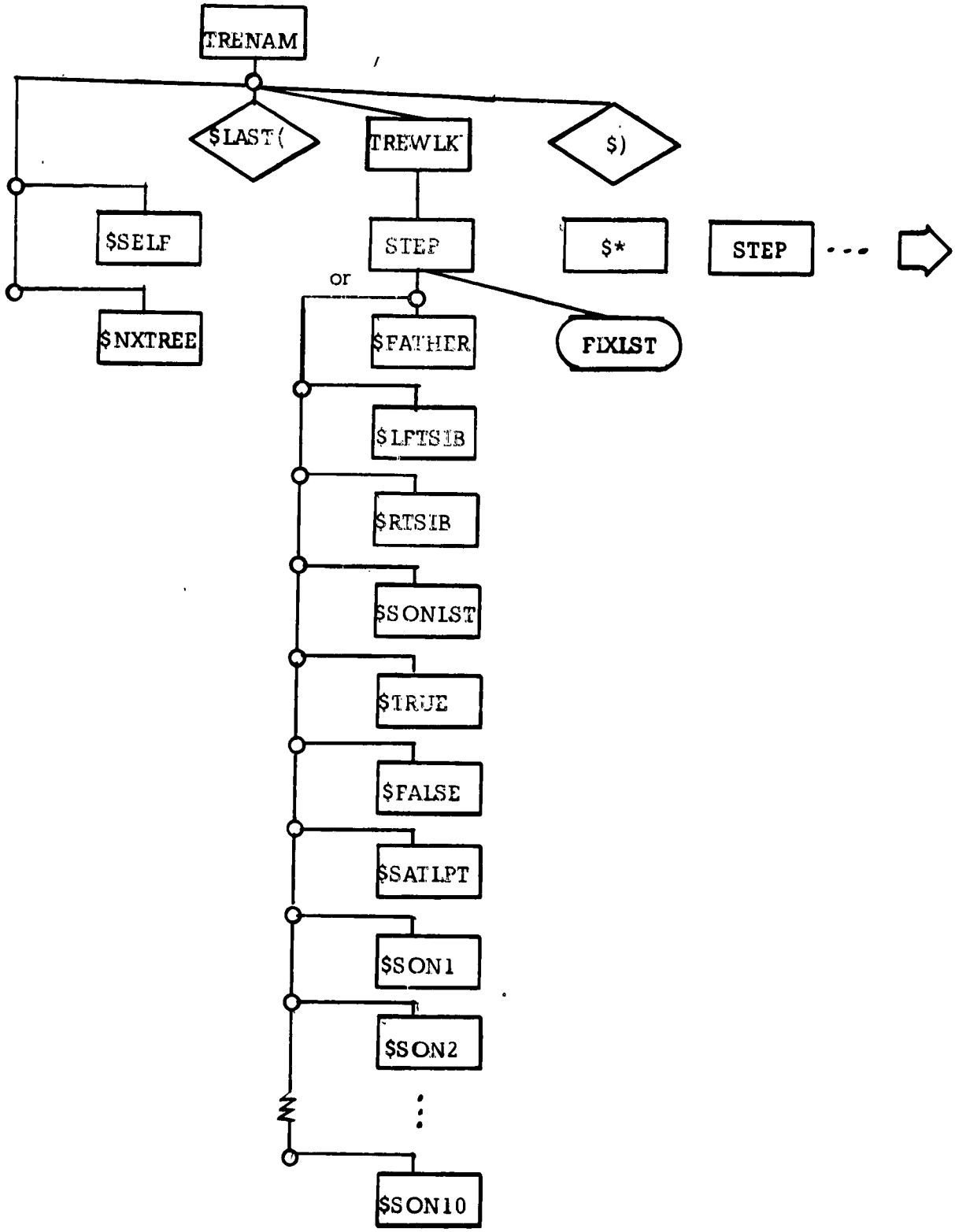

VTPTST
VTPTRE
$=
INTGER
$VTYP(
TRENAM (page6)
$)
TLUA2

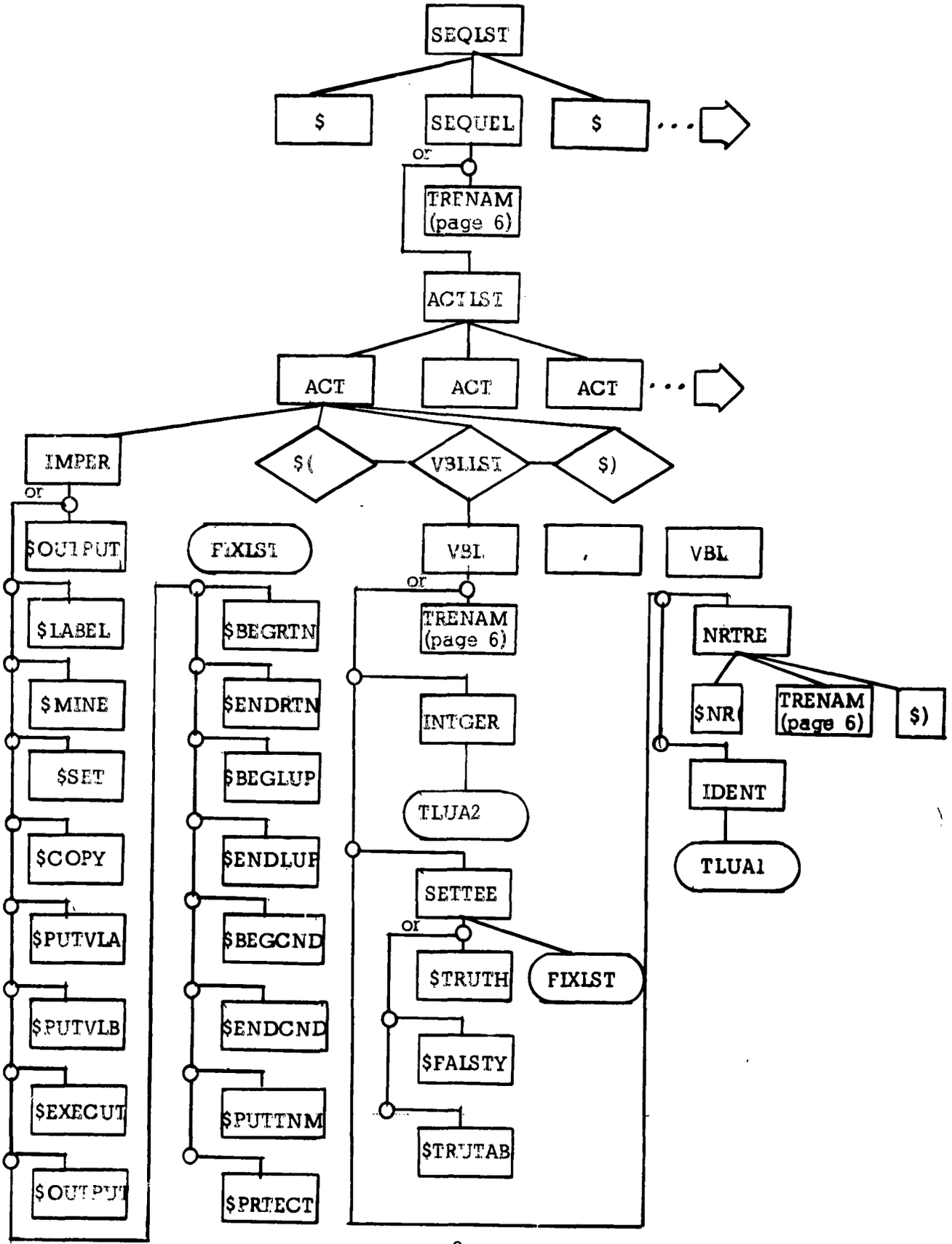

SYMTST
TRENAM (page 6)
COPULA
SYMSTG
or
$=
FIXLST
TLUA3
$IS


CPXTST
TRENAM (page 6)
$IS
HRDNES
$EASY
FIXLST
$HARD


NRTST
NRTRE
$=
INTGER
$NR(
TRENAM (page 6)
$)
TLUA2


SYNTST
TRENAM (page 6)
COPULA
SYNTYP
or
$=
FIXLST
TLUA1
$IS


TRENAM
$LAST(
TREWLK
$)
$SELF
STEP
$*
STEP
or
$NXTREE
$FATHER
FIXLST
$LFTSIB
$RTSIB
$SONLST
$TRUE
$FALSE
$SATLPT
$SON1
$SON2
$SON10


SEQLST
$
SEQUEL
$
or
TRENAM (page 6)
ACTLST
ACT
ACT
ACT
IMPER
$(
V3LIST
$)
or
$OUTPUT
$LABEL
$MINE
$SET
$COPY
$PUTVLA
$PUTVLB
$EXECUT
$OUTPUT
FIXLST
$BEGRTN
$ENDRTN
$BEGLUP
$ENDLUP
$BEGCND
$ENDCND
$PUTTNM
$PRTECT
VBL
,
VBL
or
TRENAM (page 6)
INTGER
TLUA2
SETTEE
or
$TRUTH
FIXLST
$FALSTY
$TRUTAB
NRTRE
$NR(
TRENAM (page 6)
$)
IDENT
TLUA1

Appendix II

The technique described in this report has been tested experimentally be defining the translation of the algebraic language L_o* (into IBM 7090) in BNFGSL and bootstrapping an L_o Translater into existence. To give some notion of the appearance of the BNFGSL for a conventional algebraic language, we append a listing of the cards which fed that bootstrap.

Obviously, the named "break-out" codes (arguments of EXECUT) and the VARITY codes used in VTYP tests are not self-explanatory, but much of the deck may be readable, and the approximate level of effort involved in preparing a new BNFGSL deck may be estimated.

---

*G. F. Leonard, "Introduction to the CL-I Programming System", with T. E. Cheatham, Jr., et al; Technical Operations, Inc., TR-59-6, January, 1960.

THIS IS A CARD LISTING BY C-E-I-R

```
IDENT=//TLLA1,INVOKE/
INTGER=//TLLA2,INVOKE/
LGRTN=CURDEC LGBODY//GENRAT/
ADDP= $+/$-//FIXLST,INVOKE/
MULOP=$*/$$///FIXLST/
RELOP=$EQ/$UE/$LT/$LE/$GR/$GE//FIXLST/
VARBLE = IDENT $( SUBSCR $) / IDENT///
SUBSCR = ASSPHR / AREX ///
AREX= TERM/ ADDP TERM/ AREX ADDP TERM///
TERM=FACTOR/TERM MULOP FACTOR///
FACTOR=VARBLE/INTGER/$( AREX $)/$( ASSPHR $)///
ASSPHR=LPTLST AREX///
LPTLST=LPT/LPTLST LPT///
LPT=VARBLE $=///
IFPHR=IFCLS UNCPHR $.///
IFCLS = $IF BOOLEX $, ///
BOOLEX=BOOTRM/BOOLEX $OR BOOTRM///
BOOTRM = BOOFAC/BOOTRM $AND BOOFAC///
BOOFAC = $NOT BOOPRM/BOOPRM///
BOOPRM = RELATN / $( BOOLEX $) / AREX ///
RELATN = AREX RELOP AREX///
UNCPHR=GOSUB/PUNPHR///
PUNPHR=SUB $./ IFSUB/PUNPHR SUB $./PUNPHR IFSUB///
SUB = IDENT $$$ PHRASE / PHRASE//ASSIGN/
PHRASE = SWPHR/EXPHR/CONPHR/BULNEG/ASSPHR/GOPHR/IDENT///
BULNEG = $NOT IDENT///
CONPHR= $CONNECT IDENT $TO IDENT///
GOSUB=GOPHR/IDENT $$$ GOPHR///
GOPHR=GOPART IDENT///
GOPART=$GOTO/$GO/$TO//FIXLST/
EXPHR=$EXIT//FIXLST/
SWPHR= $SWITCH $( AREX $, IDLIST $)///
IDLIST = IDCOMP/IDLIST IDCOMP///
IDCOMP=IDENT $,/IDENT///
IFSUB= IFPHR/ IDENT $$$ IFPHR//INHIBT/
STATE = IFSUB/DECLAR/SUB $.//GENRAT/
LOCODE=STATE/LOCODE STATE///
LGBODY = $BGNLG$/ LOCODE $$/ENDLG///
CURDEC = $ROUTIN IDENT//METAGN/
DECLAR = TBLITM / TBLDEC / RGNDCL / RTNHED // GENRAT /
NULL = //NULTYP/
GLOBPH = $GLOBAL $( INTGER $) /$GLOPAI///
ALOSTF = $LOCAL/GLOBPH/NULL///
RTNHED = IDENT $RTN ALOSTF $$$//METAGN/
TBLHED = $TABLE IDENT INTGER ALOSTF $$$ $BEGIN//METAGN/
TBLITM = $ITEM IDENT INTGER INTGER INTGER $$$//METAGN/
TBLITL = TBLITM $END/TBLITM TBLITL///
TBLDEC = TBLHED TBLITL///
RGNDCL = $BEGRGN / $ENDRGN // FIXLST, METAGN /
/END
CSI..
IF SUBSCR, $SONI $PUTVLA(NR(LFTSIB*LFTSIP))
    EXECUT(INITPI) PUTVLA(MPYDIV) PUTVLB(1)
    PUTVLB(SCNI) EXECUT(DVDCON) OUTPUT EXECUT(RMACER)
```

```
    OUTPUT $LFTSIB $PUTVLB(SELF) PUTVLB(LFTSIB+LFTSIB)
    $LFTSIB $EXECUT(FRBICN) EXECUT(NRBICN).
IF STATE ,$SCN1.
IF IFSLB AND SCN2=0,$SCN1.
IF IFSLB,$LABEL(SCN1) $SCN3.
IF IFPHR AND NR(SCN1*SCN2) = 1 AND SCN1*SCN2*SCN1*SCN2 X= 0,
  $ SCN1 $ SCN2*SCN1 $ ENDCND .
IF IFPHR,$SCN1 BEGCND $SCN2*SCN1 $ENDCND.
IF BCCTRM, $SET(TRUTH,RTSIB*SCN1) SET(FALSTY,
    FATHER*FATHER*RTSIB) $SCN2.
IF BCCTRM,$COPY(TRUTAB,FATHER) $SCN1.
IF BCCTRM AND LFTSIB=0 AND RTSIB=0,
    $COPY(TRUTAB,FATHER) $SCN1.
IF BCCTRM AND LFTSIB=0,$COPY(TRUTH,FATHER)
    SET(FALSTY,RTSIB*RTSIB) $SCN1 $BEGCND
    $RTSIB*RTSIB.
IF BCCTRM AND RTSIB=0,$COPY(TRUTAB,FATHER)
    $SCN1 $ENDCND.
IF BCCTRM,$COPY(TRUTH,FATHER) SET(FALSTY,
    RTSIB*RTSIB) $SCN1 $RTSIB*RTSIB.
IF BCCFAC AND LFTSIB=0 AND RTSIB=0,
    $COPY(TRUTAB,FATHER) $SCN2 $SCN1.
IF BCCFAC AND LFTSIB=0,$COPY(FALSTY,FATHER)
    SET(TRUTH,RTSIB*RTSIB) $SCN2$SCN1$BEGCND
    $RTSIB*RTSIB.
IF BCCFAC AND RTSIB = 0 AND NR(FATHER*FATHER) = 1
  AND FATHER*FATHER*FATHER IS IFCLS,
   $ COPY(TRUTAB, FATHER) $ SCN2 $ SCN1 .
IF BCCFAC AND RTSIB=0,$COPY(TRUTAB,FATHER)
    $SCN2 $SCN1 $ENDCND.
IF BCCFAC,$SET(TRUTH,RTSIB*RTSIB) COPY(FALSTY,
    FATHER) $SCN2 $SCN1 $RTSIB*RTSIB.
IF BCCPRM AND LFTSIB=0,$COPY(TRUTAB,FATHER)
    $SCN1.
IF BCCPRM, $COPY(TRUTH,FATHER*FALSE)
    COPY(FALSTY,FATHER*TRUE) $SCN1.
    $ATH* $SCN1 $SCN3 $OUTPUT(RELATE,NR($SCN2),$SCN1,$SCN3,
    FATHER*TRUE,FATHER*FALSE).
IF AREX AND SELF IS HARD AND FATHER IS BCCPRM , $SCN1 $ MINE
  $OUTPUT (RELATE, 1, SELF, 0, FATHER*TRUE, FATHER*FALSE) .
IF AREX AND SELF IS HARD,$SON1 $MINE.
IF AREX AND FATHER IS BCCPRM , $ SCN1
    $OUTPUT (RELATE, 1, SCN1, 0, FATHER*TRUE,FATHER*FALSE).
IF ARTX, $SCN1.
IF RUNPHR,$SON1.
IF SUB AND SCN2=0,$SCN1*SON1 $RTSIB*RTSIB.
IF SUB,$LABEL(SCN1) $SON3*SON1 $RTSIB*RTSIB.
IF SWPHR,$SCN3 $OUTPUT(SWITCH,SCN3,NR(SCN5))
    $SCN5*SCN1.
IF IOCOMP,$OUTPUT(SWIRA,SON1) $RTSIB.
IF EXPHR, $BEGRGN OUTPUT(EXIT) ENDRGN.
IF CONPHR, $OUTPUT(CONNECT,SCN2,SCN4
IF GOPHR,$OUTPUT(TRANS,SON2).
IF GOSLB AND SCN2 X= 0, $LABEL(SCN1) $SCN3.
IF GOSLB, $SON1.
IF ASSPHR, $SON2 $MINE $SON1
IF LPTLST,$SCN1.
IF LPT, $SCN1 $RTSIB.
IF VARELE AND SON3=0 AND VTYP(SCN1)=3 AND
    FATHER IS LPT, $OUTPUT(STORE,SCN1,FATHER*FATHER*RTSIB).
IF VARELE AND SON3=0 AND VTYP(SCN1)=4 AND
```

```
    [illegible] AND SON3=0 AND VTYP(SON1)=4,
    [illegible] $PUTVLB(1) $SON1 $PUTVLA(PUT) $SON1
    [illegible]VLB(FATHER*FATHER*RTSIB) $SON1.
    [illegible] AND SON3 X=0, $SON3 $OUTPOT.
IF VARBLE AND FATHER IS LPT, $OUTPUT(STORE,SON1,
    FATHER*FATHER*RTSIB).
IF $( AND RTSIB IS SUBSCR AND FATHER*FATHER IS FACTOR,
    $PUTVLB(0) $RTSIB $PUTVLA(GET) $RTSIB $RTSIB.
IF $( AND RTSIB IS SUBSCR,
    $PUTVLB(1) $RTSIB $PUTVLA(PUT) $RTSIB
    $PUTVLB(FATHER*FATHER*FATHER*RTSIB) $RTSIB.
IF $) AND VTYP(LFTSIB*LFTSIB*LFTSIB)=3,
    $PUTVLB(0) EXECUT(ADTVCN) $LFTSIB.
IF $) AND VTYP(LFTSIB*LFTSIB*LFTSIB)=4,
    $PUTVLB(1) PUTVLB(0) EXECUT(ADTVCN) $LFTSIB.
IF $( ,$RTSIB.
IF TERM AND SELF IS HARD,$SON1 $MINE $LFTSIB
    $RTSIB*RTSIB.
IF TERM,$SON1 $LFTSIB $RTSIB*RTSIB.
IF ADOP AND LFTSIB=0, $RTSIB $OUTPUT(SIGN,NR(SELF),RTSIB)
  $RTSIB*RTSIB*RTSIB.
IF ADOP,$OUTPUT(ADDSUB,NR(SELF),LAST(LFTSIB*LFTSIB),
    RTSIB) $RTSIB*RTSIB*RTSIB.
IF FACTOR AND SELF IS HARD,$SON1 $MINE
    $LFTSIB $RTSIB*RTSIB.
IF FACTOR,$SON1 $LFTSIB $RTSIB*RTSIB.
IF MULOP,$OUTPUT(MPYDIV,NR(SELF),LAST(LFTSIB*
    LFTSIB),RTSIB) $RTSIB*RTSIB*RTSIB.
IF IDENT AND VTYP(SELF)=7, $OUTPUT(FUNC,SELF).
IF IDENT AND FATHER IS BOOPRM, $OUTPUT(RELATE,1,
    SELF,0,FATHER*TRUE,FATHER*FALSE).
IF IDENT AND FATHER IS VARBLE, $PUTVLA(SELF) EXECUT(INITPT)
    PUTVLA(LOCATE) $FATHER $EXECUT(HOSKYP) PUTVLB(1)
    PUTVLB(0) EXECUT(ADTVCN) OUTPOT $FATHER $PUTVLB(SELF)
```

```
    $EXECUT(ADDCON) $FATHER $EXECUT(FRBTCN) EXECUT(NRBTCN)
    OUTPUT.
IF ODERT, $OUTPUT(STORE,SELF,1).
IF BULNEG, $OUTPUT(STORE,SON2,0).
IF LCRTN, $ENDRTN.
IF CURDEC, $PUTVLA(NR(SON2)) EXECUT(MYRTDC) BEGRTN.
IF TBLITM,$PUTVLA(NR(SON2)) PUTVLB(NR(SON3))
    PUTVLB(NR(SON4)) PUTVLB(NR(SON5)) EXECUT(TBLITDC).
IF TBLHED AND SON4*SON1 = GLOBPH,
    $PUTVLA(NR(SON2)) PUTVLB(6) PUTVLB(NR[illegible]
    PUTVLB(NR(SON4*SON1*SON3)) EXECUT(TABLDC).
IF TBLHED,
    $PUTVLA(NR(SON2)) PUTVLB(8) PUTVLB(NR(SON3))
    PUTVLB(0) EXECUT(TABLDC).
IF RGNDCL AND NR(SELF) = 0 , $ BEGRGN .
IF RGNDCL , $ ENDRGN .
IF RTNHED AND SON3*SON1 = GLOBPH,
    $PUTVLA(NR(SON1)) PUTVLB(2)
    PUTVLB(NR(SON3*SON1*SON3)) EXECUT(RTNDEC).
IF RTNHED,
    $PUTVLA(NR(SON1)) PUTVLB(1) PUTVLB(0) EXECUT(RTNDEC).
/GSL
```

YOUR CARD TOTAL IS --- 181